**NoSQL databases**

NoSQL databases store and retrieve data in a format other than traditional SQL relational tables. They are designed to handle large volumes of unstructured or semi-structured data. As such they typically have fewer relational constraints and consistency checks than SQL, and claim significant benefits in terms of scalability, flexibility, and performance.

Like SQL databases, users interact with data in NoSQL databases using queries that are passed by the application to the database. However, different NoSQL databases use a wide range of query languages instead of a universal standard like SQL (Structured Query Language). This may be a custom query language or a common language like XML or JSON.

**NoSQL database models**

There is a wide variety of NoSQL databases. In order to detect vulnerabilities in a NoSQL database, it helps to understand the model framework and language.

Some common types of NoSQL databases include:

* Document stores - These store data in flexible, semi-structured documents. They typically use formats such as JSON, BSON, and XML, and are queried in an API or query language. Examples include MongoDB and Couchbase.
* Key-value stores - These store data in a key-value format. Each data field is associated with a unique key string. Values are retrieved based on the unique key. Examples include Redis and Amazon DynamoDB.
* Wide-column stores - These organize related data into flexible column families rather than traditional rows. Examples include Apache Cassandra and Apache HBase.
* Graph databases - These use nodes to store data entities, and edges to store relationships between entities. Examples include Neo4j and Amazon Neptune.

**NoSQL injection**

NoSQL injection is a vulnerability where an attacker is able to interfere with the queries that an application makes to a NoSQL database. NoSQL injection may enable an attacker to:

* Bypass authentication or protection mechanisms.
* Extract or edit data.
* Cause a denial of service.
* Execute code on the server.

NoSQL databases store and retrieve data in a format other than traditional SQL relational tables. They use a wide range of query languages instead of a universal standard like SQL, and have fewer relational constraints.

There are two different types of NoSQL injection:

* Syntax injection - This occurs when you can break the NoSQL query syntax, enabling you to inject your own payload. The methodology is similar to that used in SQL injection. However, the nature of the attack varies significantly, as NoSQL databases use a range of query languages, types of query syntax, and different data structures.
* Operator injection - This occurs when you can use NoSQL query operators to manipulate queries.

In this topic, we'll look at how to test for NoSQL vulnerabilities in general, then focus on exploiting vulnerabilities in MongoDB, which is the most popular NoSQL database. We've also provided some labs so you can practice what you've learned.

**NoSQL syntax injection**

You can potentially detect NoSQL injection vulnerabilities by attempting to break the query syntax. To do this, systematically test each input by submitting fuzz strings and special characters that trigger a database error or some other detectable behavior if they're not adequately sanitized or filtered by the application.

If you know the API language of the target database, use special characters and fuzz strings that are relevant to that language. Otherwise, use a variety of fuzz strings to target multiple API languages.

**Detecting syntax injection in MongoDB**

Consider a shopping application that displays products in different categories. When the user selects the **Fizzy drinks** category, their browser requests the following URL:

https://insecure-website.com/product/lookup?category=fizzy

This causes the application to send a JSON query to retrieve relevant products from the product collection in the MongoDB database:

this.category == 'fizzy'

To test whether the input may be vulnerable, submit a fuzz string in the value of the category parameter. An example string for MongoDB is:

'"`{

;$Foo}

$Foo \xYZ

Use this fuzz string to construct the following attack:

https://insecure-website.com/product/lookup?category='%22%60%7b%0d%0a%3b%24Foo%7d%0d%0a%24Foo%20%5cxYZ%00

If this causes a change from the original response, this may indicate that user input isn't filtered or sanitized correctly.

**Note**

NoSQL injection vulnerabilities can occur in a variety of contexts, and you need to adapt your fuzz strings accordingly. Otherwise, you may simply trigger validation errors that mean the application never executes your query.

In this example, we're injecting the fuzz string via the URL, so the string is URL-encoded. In some applications, you may need to inject your payload via a JSON property instead. In this case, this payload would become '\"`{\r;$Foo}\n$Foo \\xYZ\u0000.

**Determining which characters are processed**

To determine which characters are interpreted as syntax by the application, you can inject individual characters. For example, you could submit ', which results in the following MongoDB query:

this.category == '''

If this causes a change from the original response, this may indicate that the ' character has broken the query syntax and caused a syntax error. You can confirm this by submitting a valid query string in the input, for example by escaping the quote:

this.category == '\''

If this doesn't cause a syntax error, this may mean that the application is vulnerable to an injection attack.

**Confirming conditional behavior**

After detecting a vulnerability, the next step is to determine whether you can influence boolean conditions using NoSQL syntax.

To test this, send two requests, one with a false condition and one with a true condition. For example you could use the conditional statements ' && 0 && 'x and ' && 1 && 'x as follows:

https://insecure-website.com/product/lookup?category=fizzy'+%26%26+0+%26%26+'xhttps://insecure-website.com/product/lookup?category=fizzy'+%26%26+1+%26%26+'x

If the application behaves differently, this suggests that the false condition impacts the query logic, but the true condition doesn't. This indicates that injecting this style of syntax impacts a server-side query.

**Overriding existing conditions**

Now that you have identified that you can influence boolean conditions, you can attempt to override existing conditions to exploit the vulnerability. For example, you can inject a JavaScript condition that always evaluates to true, such as '||1||':

https://insecure-website.com/product/lookup?category=fizzy%27%7c%7c%31%7c%7c%27

This results in the following MongoDB query:

this.category == 'fizzy'||'1'=='1'

As the injected condition is always true, the modified query returns all items. This enables you to view all the products in any category, including hidden or unknown categories.

**Warning**

Take care when injecting a condition that always evaluates to true into a NoSQL query. Although this may be harmless in the initial context you're injecting into, it's common for applications to use data from a single request in multiple different queries. If an application uses it when updating or deleting data, for example, this can result in accidental data loss.

You could also add a null character after the category value. MongoDB may ignore all characters after a null character. This means that any additional conditions on the MongoDB query are ignored. For example, the query may have an additional this.released restriction:

this.category == 'fizzy' && this.released == 1

The restriction this.released == 1 is used to only show products that are released. For unreleased products, presumably this.released == 0.

In this case, an attacker could construct an attack as follows:

https://insecure-website.com/product/lookup?category=fizzy'%00

This results in the following NoSQL query:

this.category == 'fizzy'\u0000' && this.released == 1

If MongoDB ignores all characters after the null character, this removes the requirement for the released field to be set to 1. As a result, all products in the fizzy category are displayed, including unreleased products.

**NoSQL operator injection**

NoSQL databases often use query operators, which provide ways to specify conditions that data must meet to be included in the query result. Examples of MongoDB query operators include:

* $where - Matches documents that satisfy a JavaScript expression.
* $ne - Matches all values that are not equal to a specified value.
* $in - Matches all of the values specified in an array.
* $regex - Selects documents where values match a specified regular expression.

You may be able to inject query operators to manipulate NoSQL queries. To do this, systematically submit different operators into a range of user inputs, then review the responses for error messages or other changes.

**Submitting query operators**

In JSON messages, you can insert query operators as nested objects. For example, {"username":"wiener"} becomes {"username":{"$ne":"invalid"}}.

For URL-based inputs, you can insert query operators via URL parameters. For example, username=wiener becomes username[$ne]=invalid. If this doesn't work, you can try the following:

1. Convert the request method from GET to POST.
2. Change the Content-Type header to application/json.
3. Add JSON to the message body.
4. Inject query operators in the JSON.

**Note**

You can use the Content Type Converter extension to automatically convert the request method and change a URL-encoded POST request to JSON.

**Detecting operator injection in MongoDB**

Consider a vulnerable application that accepts a username and password in the body of a POST request:

{"username":"wiener","password":"peter"}

Test each input with a range of operators. For example, to test whether the username input processes the query operator, you could try the following injection:

{"username":{"$ne":"invalid"},"password":{"peter"}}

If the $ne operator is applied, this queries all users where the username is not equal to invalid.

If both the username and password inputs process the operator, it may be possible to bypass authentication using the following payload:

{"username":{"$ne":"invalid"},"password":{"$ne":"invalid"}}

This query returns all login credentials where both the username and password are not equal to invalid. As a result, you're logged into the application as the first user in the collection.

To target an account, you can construct a payload that includes a known username, or a username that you've guessed. For example:

{"username":{"$in":["admin","administrator","superadmin"]},"password":{"$ne":""}}

**Exploiting syntax injection to extract data**

In many NoSQL databases, some query operators or functions can run limited JavaScript code, such as MongoDB's $where operator and mapReduce() function. This means that, if a vulnerable application uses these operators or functions, the database may evaluate the JavaScript as part of the query. You may therefore be able to use JavaScript functions to extract data from the database.

**Exfiltrating data in MongoDB**

Consider a vulnerable application that allows users to look up other registered usernames and displays their role. This triggers a request to the URL:

https://insecure-website.com/user/lookup?username=admin

This results in the following NoSQL query of the users collection:

{"$where":"this.username == 'admin'"}

As the query uses the $where operator, you can attempt to inject JavaScript functions into this query so that it returns sensitive data. For example, you could send the following payload:

admin' && this.password[0] == 'a' || 'a'=='b

This returns the first character of the user's password string, enabling you to extract the password character by character.

You could also use the JavaScript match() function to extract information. For example, the following payload enables you to identify whether the password contains digits:

admin' && this.password.match(/\d/) || 'a'=='b

**Identifying field names**

Because MongoDB handles semi-structured data that doesn't require a fixed schema, you may need to identify valid fields in the collection before you can extract data using JavaScript injection.

For example, to identify whether the MongoDB database contains a password field, you could submit the following payload:

https://insecure-website.com/user/lookup?username=admin'+%26%26+this.password!%3d'

Send the payload again for an existing field and for a field that doesn't exist. In this example, you know that the username field exists, so you could send the following payloads:

admin' && this.username!=' admin' && this.foo!='

If the password field exists, you'd expect the response to be identical to the response for the existing field (username), but different to the response for the field that doesn't exist (foo).

If you want to test different field names, you could perform a dictionary attack, by using a wordlist to cycle through different potential field names.

**Note**

You can alternatively use NoSQL operator injection to extract field names character by character. This enables you to identify field names without having to guess or perform a dictionary attack. We'll teach you how to do this in the next section.

**Exploiting NoSQL operator injection to extract data**

Even if the original query doesn't use any operators that enable you to run arbitrary JavaScript, you may be able to inject one of these operators yourself. You can then use boolean conditions to determine whether the application executes any JavaScript that you inject via this operator.

**Injecting operators in MongoDB**

Consider a vulnerable application that accepts username and password in the body of a POST request:

{"username":"wiener","password":"peter"}

To test whether you can inject operators, you could try adding the $where operator as an additional parameter, then send one request where the condition evaluates to false, and another that evaluates to true. For example:

{"username":"wiener","password":"peter", "$where":"0"}{"username":"wiener","password":"peter", "$where":"1"}

If there is a difference between the responses, this may indicate that the JavaScript expression in the $where clause is being evaluated.

**Extracting field names**

If you have injected an operator that enables you to run JavaScript, you may be able to use the keys() method to extract the name of data fields. For example, you could submit the following payload:

"$where":"Object.keys(this)[0].match('^.{0}a.\*')"

This inspects the first data field in the user object and returns the first character of the field name. This enables you to extract the field name character by character.

**Exfiltrating data using operators**

Alternatively, you may be able to extract data using operators that don't enable you to run JavaScript. For example, you may be able to use the $regex operator to extract data character by character.

Consider a vulnerable application that accepts a username and password in the body of a POST request. For example:

{"username":"myuser","password":"mypass"}

You could start by testing whether the $regex operator is processed as follows:

{"username":"admin","password":{"$regex":"^.\*"}}

If the response to this request is different to the one you receive when you submit an incorrect password, this indicates that the application may be vulnerable. You can use the $regex operator to extract data character by character. For example, the following payload checks whether the password begins with an a:

{"username":"admin","password":{"$regex":"^a\*"}}

**Timing based injection**

Sometimes triggering a database error doesn't cause a difference in the application's response. In this situation, you may still be able to detect and exploit the vulnerability by using JavaScript injection to trigger a conditional time delay.

To conduct timing-based NoSQL injection:

1. Load the page several times to determine a baseline loading time.
2. Insert a timing based payload into the input. A timing based payload causes an intentional delay in the response when executed. For example, {"$where": "sleep(5000)"} causes an intentional delay of 5000 ms on successful injection.
3. Identify whether the response loads more slowly. This indicates a successful injection.

The following timing based payloads will trigger a time delay if the password beings with the letter a:

admin'+function(x){var waitTill = new Date(new Date().getTime() + 5000);while((x.password[0]==="a") && waitTill > new Date()){};}(this)+'admin'+function(x){if(x.password[0]==="a"){sleep(5000)};}(this)+'

**Preventing NoSQL injection**

The appropriate way to prevent NoSQL injection attacks depends on the specific NoSQL technology in use. As such, we recommend reading the security documentation for your NoSQL database of choice. That said, the following broad guidelines will also help:

* Sanitize and validate user input, using an allowlist of accepted characters.
* Insert user input using parameterized queries instead of concatenating user input directly into the query.
* To prevent operator injection, apply an allowlist of accepted keys.